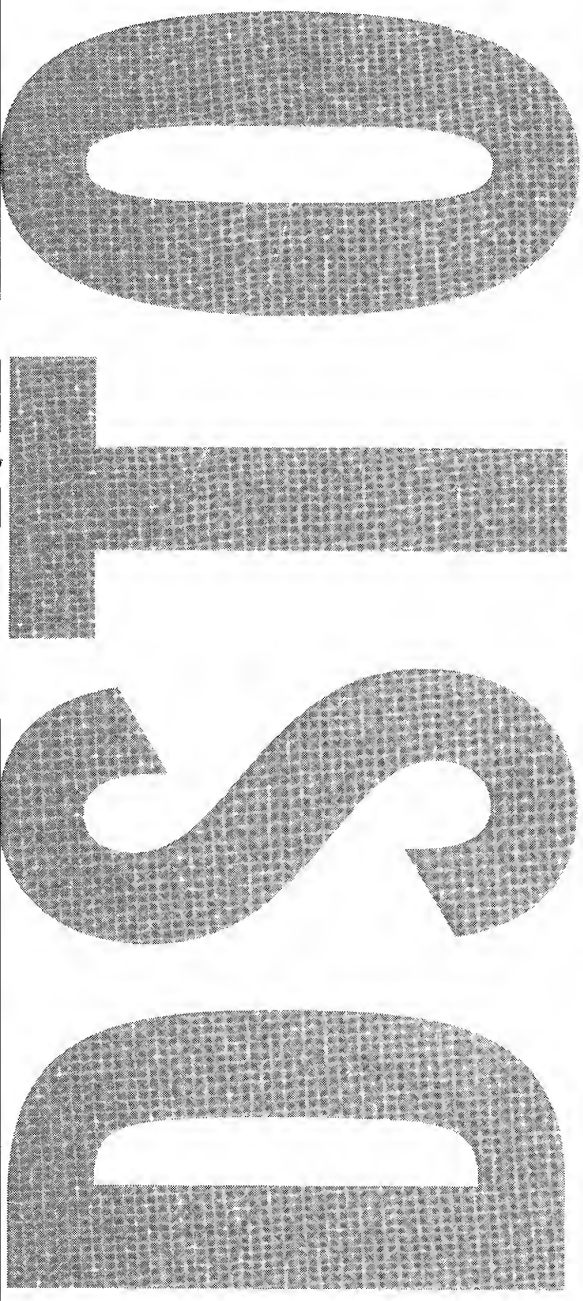
DEPARTMENT OF DEFENCE
DEFENCE SCIENCE & TECHNOLOGY ORGANISATION
DSTO

Jan 2002



Implementation of a Fast Algorithm for Segmenting SAR Imagery

David J. Robinson,
Nicholas J. Redding and
David J. Crisp

DSTO-TR-1242

Implementation of a Fast Algorithm for Segmenting SAR Imagery

David J. Robinson, Nicholas J. Redding and David J. Crisp

Surveillance Systems Division
Electronics and Surveillance Research Laboratory

DSTO–TR–1242

ABSTRACT

This report gives a detailed presentation of the implementation of a new fast algorithm for image segmentation. The original motivation for development of the algorithm was the segmentation of synthetic aperture radar (SAR) imagery into homogeneous regions for target detection in the Analysts' Detection Support System. However, the algorithm is a general one based upon Mumford-Shah functionals, and there is no technical reason why it could not also be used for other imaging modalities, including multiband imagery. The algorithm has computational complexity on the order of the Fast Fourier Transform, the benchmark for fast algorithms.

Published by

DSTO Electronics and Surveillance Research Laboratory
PO Box 1500
Edinburgh, South Australia, Australia 5111

Telephone: *(08) 8259 5555*
Facsimile: *(08) 8259 6567*

AR No. 012-076
January, 2002

APPROVED FOR PUBLIC RELEASE

Implementation of a Fast Algorithm for Segmenting SAR Imagery

EXECUTIVE SUMMARY

This report gives a detailed presentation of the implementation of a new fast algorithm for image segmentation. The original motivation for development of the algorithm was the segmentation of synthetic aperture radar (SAR) imagery into homogeneous regions for target detection in the Analysts' Detection Support System (ADSS). The ADSS was originally developed as part of Aerial Surveillance for Land Operations, Joint Project 129. It was designed to help reduce the manpower cost of the Broad Area Aerial Surveillance component of the project where the manual detection of targets in stripmap SAR is labour intensive.

The ADSS is divided into a number of modules, the first of which is called prescreening. The purpose of prescreening is to identify all the possible targets of interest in the SAR imagery with a high probability of detection. One method previously suggested to improve the performance of prescreening is to first segment the imagery into homogeneous regions using texture and then apply different prescreeners to each texture class. The rational is that each prescreener can be tuned (using prior training or otherwise) to give optimal performance on its texture class.

Segmentation is typically computationally very expensive, and segmentation for prescreening would require all the imagery collected from the SAR sensor to be segmented — a daunting task for standard algorithms. Consequently, a very fast segmentation algorithm called the *full λ-schedule* was developed with prescreening in mind. The basic principles of this algorithm have been presented elsewhere. In this report we describe an implementation of the algorithm. We include full details of the data structures, procedures and functions used.

The performance of our initial implementations did not reflect our theoretical estimates. However, as described in this report, significant recent enhancements of the data structures by the third author have over come this problem. We believe that we now have the fastest possible implementation of the full λ-schedule algorithm. Computational complexity calculations and numerical evidence, both presented in the report, show that the complexity is of order $\mathcal{O}(n \log_2 n)$ where n is the number of image pixels. This complexity is the benchmark for fast algorithms.

Future work will involve testing the utility of the algorithm for target detection and extending the class of image models it uses. The initial steps in both these directions have already been taken. The algorithm has been incorporated into the ADSS software suite and once the coding of the entire suite has been completed and suitable data is available we will be able to test its target detection utility. On the second point, the algorithm described here only allows piecewise constant image models. This means some of the more subtle features of the SAR imagery cannot be exploited. However, progress has since been made on expanding the class of applicable image models to include piecewise polynomial models. It is anticipated that implementing these models will only involve a few local

changes to the segmentation code and that these changes will not effect the computational complexity of the algorithm.

Authors

David J. Robinson
Vision Abell Pty Ltd

David Robinson received a B.Sc. in applied mathematics and computer science and B.E. in electrical and electronic engineering all from the University of Adelaide, in 1995 and 1996, respectively. He joined the Electronic Warfare Division of the Defence Science and Technology Organisation (DSTO) in Adelaide as a Professional Officer in 1996 where he investigated and modelled the tracking modes of airborne Pulsed Doppler radars. In 1998 he was appointed as a software engineer with Vision Abell Pty Ltd, where he programmed real time embedded systems. During 1999 he was under contract to Surveillance Systems Division of DSTO to assist in the evaluation and development of algorithms for the Analysts' Detection Support System (ADSS).

Nicholas J. Redding
Surveillance Systems Division

Nicholas Redding received a B.E. and Ph.D. in electrical engineering all from the University of Queensland, Brisbane, in 1986 and 1991, respectively. From 1988 he received a Research Scientist Fellowship from the Australian Defence Science and Technology Organisation (DSTO) and then joined DSTO in Adelaide as a Research Scientist after completing his Ph.D. in artificial neural networks in 1991. In 1996 he was appointed as a Senior Research Scientist in the Microwave Radar Division (now Surveillance Systems Division) of DSTO. Since joining DSTO he has applied image processing techniques to the automatic classification of ionospheric data, and more recently researched target detection (both human and algorithmic) in synthetic aperture radar imagery.

David J. Crisp
Surveillance Systems Division

David Crisp graduated from the University of Adelaide in 1987 with a B.Sc. (Hons) in Mathematics and completed his Ph.D. at the same institution in 1993. For the three years following that he held a postdoctoral research position in Mathematics at the Flinders University of South Australia. In 1997 he commenced employment as a Postdoctoral Research Fellow with the Cooperative Research Centre for Sensor, Signal and Information Processing (CSSIP). At CSSIP he worked in the Pattern Recognition Group on the application of machine learning techniques to real world problems. In September 1999 he joined the Surveillance Systems Division of the Australian Defence Science and Technology Organisation (DSTO). Since joining DSTO he has been a Research Scientist in the Image Analysis and Exploitation Group and his research has been focused on the automated detection of targets in low resolution synthetic aperture radar imagery.

Contents

Figures

Tables

1 Introduction

The Analysts' Detection Support System (ADSS) is a suite of target detection algorithms designed to reduce analysts' workload in Broad Area Aerial Surveillance (BAAS) using synthetic aperture radar imagery (SAR) [7]. The ADSS is divided into a number of modules, the first of which is called *prescreening*. The purpose of prescreening is to identify all the possible targets of interest in the SAR imagery with a high probability of detection (PD). The performance of a number of different algorithms employed in the prescreening role has been reported elsewhere [9]. In this performance analysis, all regions of the imagery were treated uniformly, irrespective of the differing characteristics that they can exhibit. One method suggested in [9] to improve the performance of prescreening is to segment the imagery into homogeneous regions using texture prior to prescreening and apply different prescreeners to each texture class. The rational is that each prescreener can be tuned (using prior training or otherwise adjusting its parameters) to give optimal performance on its texture class.

Segmentation is typically computationally very expensive, and segmentation for prescreening would require all the imagery collected from the SAR sensor to be segmented — a daunting task for standard algorithms. Consequently, a very fast segmentation algorithm called the *full λ-schedule* was developed with prescreening in mind. The basic principles of this algorithm have been presented in [8].

This report outlines what we believe to be the fastest possible implementation of the full λ-schedule algorithm. Earlier implementations of the algorithm have been produced [11], however their performance did not reflect initial estimates presented in [8]. Significant recent enhancements of the data structures by the third author have been included.

In the next section, an overview of the full λ-schedule algorithm is presented based upon [8]. Sections 3 and 4 present a detailed overview of the data structures, and the major procedures and functions, respectively, used to implement the algorithm. Section 6 presents the data and procedures required to reconstruct the segmentation at any stage.

2 Overview of the Algorithm

2.1 Philosophy

In [8] it is argued that a segmentation is a compressed description of the image, and that an unavoidable consequence of a compressed description is the introduction of some error. A good segmentation is therefore one which has a very efficient description given the associated error. One should consequently view segmentation as a compromise between the shape or character of a boundary and the fitting error in the region enclosed by the boundary.

The compromise between fitting error and shape can be presented in a rigorous mathematical framework [5] by expressing the segmentation problem with variational methods using the Mumford-Shah functionals [6]. These functionals were shown to provide a unifying framework for image segmentation. The variational framework addresses the dilemma

between shape and error by means of a parameter λ which expresses the trade-off between fitting error in a region against the region's boundary length. Koepfler *et al.* [4] call λ the "scale" parameter, but it does not parameterize the length characteristic of each region so instead we will call it the *regularisation* parameter because of the way it determines the balance between error of fit to a region and its boundary length.

The simplified form of the Mumford-Shah functionals expresses the segmentation problem as one of minimizing

$$E(u, K) = \int_{\Omega \setminus K} \|u - g\|^2 dx\, dy + \lambda \ell(K) \tag{1}$$

where Ω is the domain of the image, K is a set of segmentation boundaries with total length $\ell(K)$, g is a scalar or vector-valued function of the *channels* of the image on the domain Ω, u is a piece-wise constant approximating scalar or vector-valued function for the image which is constant over each region, and λ is the regularisation parameter. If λ is small, then a lot of boundaries are allowed and a "fine" segmentation results. As λ increases, coarser and coarser segmentations result.

The functional can be used to segment an image into its component textures by first extracting several different texture features from the original image. In this case, the channels of g are the distinct texture features and the term $||u - g||$ in (1) is defined in terms of a suitable vector space norm $||\cdot||$, see [4] for more details. Note however, that there is only one segmentation boundary and it is fitted to all image channels simultaneously. Of course, the functional can also be used to segment on the basis of grey-scale values alone. In this case, g is single channel and the term $||u - g||$ is just the absolute value of $u - g$ and so $||u - d||^2 = (u - g)^2$.

The first term of (1) can be interpreted as error in the segmentation, and the second as the length of description of the segmentation. Moreover, minimizing (1) over a range of λ is equivalent to solving the following constrained optimization over a range of ϵ:

$$\min_K \ell(K) \quad \text{subject to} \quad \int_{\Omega \setminus K} \|u - g\|^2 dx\, dy \leqslant \epsilon. \tag{2}$$

In other words, for each ϵ we find the segmentation with the shortest description that has an error of at most ϵ. Thus minimizing the Mumford-Shah functional is equivalent to finding the segmentation with shortest description for a given error (or vice-versa if we rewrite (2)) and any segmentation found by these means involves implicitly or explicitly a choice of the appropriate trade-off (*i.e.* coarseness).

2.2 Mumford-Shah Functionals

The Mumford-Shah functionals have been shown to provide a unifying framework for image segmentation [5, 4]. The method does not depend on any *a priori* knowledge of the statistics of the image and has the properties of compactness of the set of approximate solutions (which indicates that the solution set will be small), convergence of minimizing sequences of solutions, and smoothness of the locally optimal solutions. Koepfler *et al.* [4] use a special case of the Mumford-Shah functionals as defined in (1). For a piece-wise

constant approximating function, u is simply the mean value of g in the corresponding segment. (The mean is taken component-wise in the case where u and g are vector valued.) As a result, u is uniquely defined in the piece-wise constant case given the boundary K and $E(u, K)$ can be written as $E(K)$.

In order to use (1) for segmenting digital images it needs to be discretised. To do so, we now take Ω to be a set of pixels indexed by a single discrete variable $i = 1, \ldots, n$. The image g and its model u are then defined by their values $g(i)$ and $u(i)$ at each pixel. A segmentation region is a connected sub-set of Ω and a segmentation K is a partition of Ω into regions. The boundary of K is the set of pixel edges which separate the regions and its length $\ell(K)$ is the number of edges in the set. With this notation, the functional (1) becomes

$$E(u, K) = \sum_{i=1}^{n} ||u(i) - g(i)||^2 + \lambda \ell(K). \tag{3}$$

As just discussed, or fixed K, $E(u, K)$ is minimised when u is the average of g on each region. Hence we can assume u is given by

$$u(k) = \frac{1}{|O|} \sum_{i \in O} g(i) \tag{4}$$

where O is the region of K containing the pixel k and $|O|$ is its area.

Koepfler *et al.* show that minimizing the functional in (3) (to a local minimum) can be achieved using region growing. Let O_i denote the region or segment i of the image, let u_i be the average value of g on O_i and let $\partial(O_i, O_j)$ denote the common boundary of O_i and O_j, which is contained in K. Then the merging criterion is that

$$E(K \backslash \partial(O_i, O_j)) - E(K) = \frac{|O_i||O_j|}{|O_i| + |O_j|} \|u_i - u_j\|^2 - \lambda l(\partial(O_i, O_j)) \tag{5}$$

be negative, where $|\cdot|$ denotes the area of a region. The complete Koepfler *et al.* algorithm for segmentation is as follows.

Algorithm 1

1. Take the pixels of the image as the initial trivial segmentation (u_0, K_0) and $\lambda_i = \lambda_1$ as the initial regularisation parameter.

2. For each region, determine which of its adjacent regions yield the maximal energy decrease according to (5). If such a neighbouring region exists, merge the two and proceed to check the next region in the list. Continue merging until no further decrease in the energy functional is possible.

3. For every λ_i, $i = 1, \ldots, L$ calculate a segmentation by iterating step 2 above. The algorithm stops if there is just one region left or after computing a segmentation using λ_L.

There are two problems we are faced with for practical application of this algorithm. The first is selection of the λ_i values. If they are too few or far apart then the result will be a poor segmentation. Too many will be a computational burden. The second is that the order in which the regions are visited in step 2 is not specified. This lack of specific ordering makes the algorithm fast (since no searching is done for the "best" ordering) but it also means that the results will vary depending on how the list of regions is organised. These problems are addressed in the next section.

2.3 Full λ-Schedule Segmentation

The Koepfler *et al.* algorithm presented above as Algorithm 1 requires that a list of λ_i values be selected prior to running of the algorithm. This list of values is called the λ *schedule* by analogy with the temperature schedule of simulated annealing. These values determine the quality of the final segmentation, and must normally be undertaken using trial and error. An improved version of the algorithm which overcomes this problem was presented by us in [8]. It is called the *full λ-schedule* and does not require *a priori* selection of the λ *schedule*. In essence our improvement is to consider every possible (significant) value of λ_i in the λ schedule so as to achieve the best possible segmentation. However, this is done using efficient sorting algorithms and data structures so that the resulting algorithm is fast and has known computation complexity.

From (5), the decision to merge O_i and O_j occurs when $\lambda \geqslant t_{i,j}$ where $t_{i,j}$ is given by

$$t_{i,j} \equiv \frac{\frac{|O_i||O_j|}{|O_i|+|O_j|}\|u_i - u_j\|^2}{\ell(\partial(O_i, O_j))}. \tag{6}$$

In Algorithm 1, the regions are merged by scanning arbitrarily through the list of regions and selecting the best possible merge from the neighbours of each region at the current value of λ. In contrast, for the *full λ-schedule* algorithm we consider all pairs of neighbouring regions in the image and choose the best possible pair (those having the smallest value of $t_{i,j}$ from (6)) to merge. The algorithm in simplified form is as follows.

Algorithm 2

1. Take the pixels of the image as the initial trivial segmentation.
2. Of all the neighbouring pairs of regions, find the pair (O_i, O_j) that has the smallest $t_{i,j}$ from (6).
3. Merge the regions O_i and O_j to form O_{ij}.
4. Repeat the previous two steps until there is only one region, or $t_{i,j} > \lambda_{\text{stop}}$ for all pairs of neighbouring regions (O_i, O_j).

Clearly this algorithm removes the need to select a λ-schedule *a priori* but what makes it really useful is that we can implement the strategy efficiently. The first step required is to compute all the possible pairs of neighbouring regions and sort them into a list with

ascending values of t from (6). The segmentation algorithm is then a process of merging the two regions at the top of the list, say O_i and O_j, into a new region O_{ij} and then updating the list. The first update step is to determine the neighbours O_k of O_{ij} from the union of those of O_i and O_j. We must then delete from the list of t values all those that involve regions O_i or O_j. We then insert into the list at the appropriate points the $t_{ij,k}$ values that are computed from the new region O_{ij} paired with its neighbours O_k. The algorithm then repeats by merging the new top most pair of regions in the list.

Whilst Algorithm 2 is conceptually very simple, the update process just described is complicated and a fast implementation requires complex use of variable length list structures. These details are covered in the following sections.

2.3.1 Statement of Algorithm

In this section we provide a complete description of our fast implementation of Algorithm 2. The implementation is summarised in the statement of Algorithm 3 below. First however, we need to describe the data structures it relies on. The following typographic conventions will be used. We will distinguish between the mathematical entities of the algorithm and the programming constructs used to implement them using different type faces. A calligraphic font is used to designate ordered sets (lists) and their elements, and a courier font is used exclusively for programming constructs. For instance, the set of all segmentation regions $\mathcal{S}$ is implemented using the programming construct `S`. We will assume that the domain of the image Ω is rectangular and contains $m \times n$ pixels.

The algorithm operates on two primary lists, with a number of supporting list structures. The first primary list, $\mathcal{S}$, called the *region set*, keeps track of the regions and the second, $\mathcal{A}$, called the *pair set*, keeps track of neighbouring pairs of regions that have a common boundary. We discuss the region set first.

The region set $\mathcal{S} = \{\mathcal{S}_i : i = 0, \ldots, L_r - 1\}$ is an ordered list of elements $\mathcal{S}_i = (I_i, a_i, \mathbf{u}_i, \rightarrow_{\mathcal{N}_i})$ where I_i denotes the label of the i-th region, a_i denotes its area (so $a_i = |O_i|$), $\mathbf{u}_i$ denotes its grey scale vector and finally $\rightarrow_{\mathcal{N}_i}$ denotes a pointer [1] to its neighbour pair list, to be defined shortly. A subscript r indicates a quantity at iteration r of the algorithm (after r region merges have occurred) and so L_r denotes the number of regions in the domain Ω of the image. For the sake of programming efficiency, `S` implements $\mathcal{S}$ as a static array defined to have the maximum required length $L_0 = mn$. The maximum length occurs at the initialisation of the algorithm when the segmentation is the trivial one in which each pixel in Ω is a separate region. As the algorithm proceeds, more and more entries become redundant as regions are merged, but we are deliberately trading the minimal additional memory requirements against a significant gain in computational complexity so we are willing to accept this waste. (The principle of increasing memory usage to obtain greater time performance is fundamental to our development.) Because of the static nature of the `S` array, we are able to use the convention that the label I_i of the i-th region is simply the index i, the offset of the region's first pixel in the (row-ordered) image. Consequently, we drop the I_i notation for the label of the i-th region and simply

[1] The pointer is used in the programming construct `S` for the region set $\mathcal{S}$ and it is not necessary to refer to it directly in the algorithm description to follow. This is because we are able to refer mathematically to the i-th neighbour pair list directly, but when we implement it we use a pointer in each region set element.

refer to it by its index i. This detail will become clearer in the next section where we discuss implementation details.

The pair set $\mathcal{A} = \{\mathcal{A}_i : i = 0, \ldots, \alpha_r - 1\}$ is an ordered list of elements $\mathcal{A}_i = (p, q, s_{p,q}, t_{p,q}, b_{p,q})$ which are tuples of p and q, the indices (*i.e.*, labels) of the two neighbouring regions that form the pair, $s_{p,q}$, the *weighted intensity difference* between the two regions defined by

$$s_{p,q} \equiv \frac{|O_p||O_q|}{|O_p| + |O_q|} \|u_p - u_q\|^2,$$

the merge cost $t_{p,q}$ for the two regions defined by

$$t_{p,q} \equiv \frac{s_{p,q}}{l(\partial(O_p, O_q))},$$

and finally $b_{p,q} = l(\partial(O_p, O_q))$, the common boundary length between the two regions. Again for the sake of programming efficiency, the pair set $\mathcal{A}$ is implemented as a static array `A` of the maximum required length $\alpha_0 = 2mn - m - n$ that occurs in the trivial segmentation at initialisation. As segmentation proceeds, more and more of the elements of `A` are not utilised, as pairs are removed through their merging. It is never necessary to scan through the list `A` after initialisation occurs, so it is not necessary to record separately which entries do and do not contain valid information (this is also true for the construct for the region set `S`). This has the additional benefit that the index and location of $\mathcal{A}_i$ between regions p and q will not change at all during execution of the algorithm, up until the point at which the two regions have been merged. (And of course this is also true for regions in the region set construct `S`.)

The next data structure we discuss is the key to the functioning of the algorithm because it determines the order in which the regions are merged to ensure that the best merge occurs at each step. This structure, the *merge candidate list* $\mathcal{E} = \{\mathcal{E}_i : i = 0, \ldots, \gamma_r - 1\}$ is an ordered list of elements $\mathcal{E}_i = (t_i, P_i)$ which are tuples of t_i, the merge cost of the pair of regions with pair index P_i. Here γ_r denotes the length of the merge candidate list at iteration r. Note that the elements $\mathcal{E}_i$ are arranged in order of increasing t_i. In order to be able to retrieve elements from $\mathcal{E}$ for updating we need to deal with tied merge costs rigorously. We do this by using the secondary key of the associated pair index to sort tied merge costs. Specifically, if $t_i = t_j$ then we define $(t_i, P_i) < (t_j, P_j)$ if $P_i < P_j$ and $(t_i, P_i) > (t_j, P_j)$ if $P_i > P_j$.

An enhancement to the algorithm, which we will discuss later, means that it is not necessary to include every pair in the merge candidate list because a simple local test can show that most pairs do not satisfy a necessary condition for them to be the current optimal merger. For this reason, γ_r the length of $\mathcal{E}$ at iteration r will be less than α_r, the number of pairs. In fact, $\gamma_r \leqslant \frac{1}{2} L_r$, the number of regions. This enhancement makes understanding the algorithm considerably more difficult, so we postpone it until we have presented the algorithm's fundamentals and in the mean time we assume $\gamma_r = \alpha_r$.

The final type of data structure, one for each region, is an ordered list used to speed access to the relevant elements of the region and pair sets. These lists, called the *neighbouring pair (NP) lists*, contain an ordered list of the indices of a region's neighbouring regions and the pairs they form. Let $\mathcal{N}_i = \{N_{i,j} : j = 0, \ldots, \beta_{i,r} - 1\}$ denote the NP list

for region i, where $\beta_{i,r}$ denotes the number of neighbouring regions of region i at iteration r. Then the element $\mathcal{N}_{i,j} = (j, P_j)$ is a tuple of the region index j of a neighbour and the pair index P_j of the pair they form. The elements are arranged in ascending order of the region indices.

All the information necessary to compute the full λ-schedule (in particular (6)) is then stored in the above lists and it is not necessary to refer to the pixels in the image after the initialization phase (except when the segmentation is complete). Implementing Algorithm 2 then is a process of maintaining the two principal list structures in the most efficient manner possible. As part of this process, when two regions, say O_p and O_q, are merged, we assign the smaller region label, say p, to the new region O_{pq} and allow the label q to become redundant. In terms of our data structures, this means we replace the data in $\mathcal{S}_p$ for region O_p with the data for the new region O_{pq} and the data in $\mathcal{S}_q$ becomes redundant. Following such a merge, the remaining entries in the region set $\mathcal{S}$ and the corresponding entries in pair set $\mathcal{A}$ which are effected by the merge and this relabelling must then be updated accordingly. We have summarized the mechanical steps required to do this in Algorithm 3 below.

Against each step of the algorithm we have also recorded its computational complexity (unless it is $\mathcal{O}(1)$). Note that the dynamic lists $\mathcal{E}, \mathcal{N}_i$ are implemented as binary trees, which have computational complexity $\mathcal{O}(\log_2 l)$ to search, insert, and delete elements, where l is their length. Also, while the true lengths, γ_r, $\mathcal{O}(\beta_{p,r})$, etc. of these lists will vary as step $r+1$ of the algorithm is being executed we simplify the computational complexity calculations by taking them to be fixed at their maximum values for the step. We will discuss the computational complexity of the algorithm as a whole in the next section.

Algorithm 3

1. **Initialization.** The algorithm begins with the trivial segmentation K_0 in which each pixel is a separate region. The data structures are initialised as follows:

   (a) Set $r = 0$.

   (b) Initialise $\mathcal{S}$ and $\mathcal{A}$. $\mathcal{O}(mn)$

   (c) Initialise $\mathcal{E}$, which includes sorting $\mathcal{E}$ so that the merge costs t are in ascending order (the length of $\mathcal{E}$ is $\gamma_0 \leqslant \alpha_0 = 2mn - m - n$). $\mathcal{O}(mn \log_2 mn)$

   (d) For each region i, $i = 0, \ldots, mn-1$, initialise $\mathcal{N}_i$, its NP list. Initially, each region will have at most four neighbours. $\mathcal{O}(mn)$

2. **Region Merging.** Form the segmentation K_r by merging the pair corresponding to the first entry, $\mathcal{E}_0 = (t_0, P_0)$ in the merge candidate list $\mathcal{E}$ as follows:

   (a) Set $r = r + 1$.

   (b) Determine $\mathcal{E}_0 = (t_0, P_0)$ from $\mathcal{E}$. $\mathcal{O}(\log_2 \gamma_r)$

   (c) Determine the two region indices p and q from $\mathcal{A}_{P_0}$.

   (d) Determine $a_p, \mathbf{u}_p$ and $a_q, \mathbf{u}_q$ from $\mathcal{S}_p$ and $\mathcal{S}_q$, respectively.

(e) Set $a_{pq} = a_p + a_q$, and $\mathbf{u}_{pq} = (a_p\mathbf{u}_p + a_q\mathbf{u}_{pq})/a_{pq}$ and store these new values in $a_p, \mathbf{u}_p$ of $\mathcal{S}_p$ (the new region O_{pq} is replacing the old region O_p).

(f) Remove $\mathcal{N}_{p,q} = (q, P_q)$ from region p's NP list. $\mathcal{O}(\log_2 \beta_{p,r})$

(g) Remove $\mathcal{N}_{q,p} = (p, P_p)$ from region q's NP list. $\mathcal{O}(\log_2 \beta_{q,r})$

(h) For each neighbour entry $\mathcal{N}_{p,n} = (n, P_n)$, $n = 0, \ldots, \beta_{p,r} - 2$, in region p's NP list $\mathcal{N}_p$, do: $\mathcal{O}(\beta_{p,r}) \times \ldots$

i. Set $t_{p,n}$ equal to the merge cost t specified in the pair set element $\mathcal{A}_{P_n}$.

ii. Use the merge cost $t_{p,n}$ and the pair index P_n as search keys to find the pair $(t_{p,n}, P_n)$ in the merge candidate list $\mathcal{E}$ and remove it. $\mathcal{O}(\log_2 \gamma_r)$

iii. Set the weighted intensity difference s_{P_n} of the pair set element $\mathcal{A}_{P_n}$ equal to $s_{P_n} = \frac{a_{pq}a_n}{a_{pq}+a_n}\|\mathbf{u}_{pq} - \mathbf{u}_n\|^2$.

(i) For each neighbour entry $\mathcal{N}_{q,n} = (n, P_n)$, $n = 0, \ldots, \beta_{q,r} - 2$, in region q's NP list $\mathcal{N}_q$, do $\mathcal{O}(\beta_{q,r}) \times \ldots$

i. Set $t_{q,n}$ equal to the merge cost t specified in the pair set element $\mathcal{A}_{P_n}$.

ii. Use the merge cost $t_{q,n}$ and the pair index P_n as search keys to find the pair $(t_{q,n}, P_n)$ in the merge candidate list $\mathcal{E}$ and remove it. $\mathcal{O}(\log_2 \gamma_r)$

iii. Check *whether or not* the region with index n already exists in region p's NP list $\mathcal{N}_p$ $\mathcal{O}(\log_2 \beta_{p,r})$

If not then

A. Insert a new entry $\mathcal{N}_{p,n} = (n, P_n)$ into $\mathcal{N}_p$. $\mathcal{O}(\log_2 \beta_{p,r})$

B. Change whichever of the two region indices p_{P_n} and q_{P_n} from $\mathcal{A}_{P_n} = (p_{P_n}, q_{P_n}, s_{P_n}, t_{P_n}, b_{P_n})$ that is equal to q to be equal to p instead. If necessary, swap the labels p_{P_n} and q_{P_n} in $\mathcal{A}_{P_n}$ to ensure that $p_{P_n} < q_{P_n}$.

C. Set the weighted intensity difference s_{P_n} of the pair set element $\mathcal{A}_{P_n}$ equal to $s_{P_n} = \frac{a_{pq}a_n}{a_{pq}+a_n}\|\mathbf{u}_{pq} - \mathbf{u}_n\|^2$.

D. Remove the old entry $\mathcal{N}_{n,q}$ from $\mathcal{N}_n$ and insert the new entry $\mathcal{N}_{n,p} = (p, P_n)$. $\mathcal{O}(\log_2 \beta_{n,r})$

else

A. Increment b_{P_n} from $\mathcal{A}_{P_n}$ by b_{P_q} from $\mathcal{A}_{P_q}$.

B. Remove the old entry $\mathcal{N}_{n,q}$ from $\mathcal{N}_n$. $\mathcal{O}(\log_2 \beta_{n,r})$

(j) For each entry $\mathcal{N}_{p,n} = (n, P_n)$ of $\mathcal{N}_p$ do $\mathcal{O}(\beta_{p,r+1}) \times \ldots$

i. Set $t_{P_n} = \frac{s_{P_n}}{b_{P_n}}$ in $\mathcal{A}_{P_n}$.

ii. Insert the element $\mathcal{E}_j = (t_{P_n}, P_n)$ into $\mathcal{E}$ at the appropriate point. $\mathcal{O}(\log_2 \gamma_r)$

3. **Loop.** Repeat step 2 until only one region is left or $t_0 > \lambda_{\text{stop}}$, where $\mathcal{E}_0 = (t_0, P_0)$ is the first entry in $\mathcal{E}$. $\mathcal{O}(mn) \times \ldots$

2.3.2 Computational Complexity

In determining the computational complexity of the above algorithm, the first detail to note is that for an image of $m \times n$ pixels, there are $L_0 = mn$ regions in the initial trivial segmentation, corresponding to $\alpha_0 = 2mn - m - n$ pairs of neighbouring regions. Secondly, because each step of the algorithm reduces the number of regions by one, the algorithm

will run for a maximum of $mn - 1$ steps. Consequently the computational complexity of the algorithm is going to be either the initialisation cost of $\mathcal{O}(mn \log_2 mn)$, or $mn - 1$ times the cost of the most complex operations in Step 2.

The deepest loops in step 2 have a computational cost of $\mathcal{O}(\log_2 \gamma_r)$, $\mathcal{O}(\log_2 \beta_{p,r})$, and $\mathcal{O}(\log_2 \beta_{n,r})$, and these steps are repeated either $\beta_{p,r}$ or $\beta_{q,r}$ times. Thus adding all contributions of all sub-steps (and ignoring repeated terms) we have a complexity of

$$\mathcal{O}(\log_2 \gamma_r + \log_2 \beta_{p,r} + \log_2 \beta_{q,r} + \beta_{p,r} \log_2 \gamma_r + \beta_{q,r}(\log_2 \gamma_r + \log_2 \beta_{p,r} + \log_2 \beta_{q,r}) + \beta_{p,r+1})$$

for Step 2. Obviously we can simplify this by using an upper bound on the number of neighbours a region can have. Let us denote the maximum number of neighbouring regions by $\beta_{\max}$ so that $\beta_{i,r} \leqslant \beta_{\max}$ for all regions i. We can now re-write the computational complexity as

$$\mathcal{O}(\beta_{\max} \log_2 \gamma_r + \beta_{\max} \log_2 \beta_{\max}). \tag{7}$$

An upper bound on $\beta_{\max}$ is $\beta_{\max} \leqslant L_r \leqslant mn$ since L_r is the number of regions at iteration r. However, it seems possible that we can do better. Our algorithm produces a *2-normal segmentation* at every step (the segmentation at each step is a minimum of (1) for the current λ value). Consequently each segmentation obeys the isoperimetric and inverse isoperimetric inequalities [4]. It follows that there may be an upper bound which is independent of the size of image and the algorithm step number. Unfortunately, we have not been able to find such an upper bound.

Using the upper bound $\beta_{\max} \leqslant L_r$, it follows from (7) that the complexity of Step 2 is no worse than

$$\mathcal{O}(L_r \log_2 \gamma_r + L_r \log_2 L_r).$$

The complexity of the Step 3 is simply the addition of contributions from Step 2 and so the complexity of Step 2 and Step 3 combined is no worse than

$$\sum_{r=1}^{mn-1} \mathcal{O}(L_r \log_2 \gamma_r + L_r \log_2 L_r) = \mathcal{O}((mn)^2 \log_2 mn) \tag{8}$$

where we have used the facts that $\gamma_r < 2mn - m - n - r$ (since at least one pair of regions is removed at each step of the algorithm) and $L_r = mn - r$ to derive the last equality. Clearly, this last estimate is an upper bound on the complexity of the algorithm as a whole since the initialisation step has complexity $\mathcal{O}(mn \log_2 mn)$.

If on the other hand $\beta_{\max}$ is independent of the size of the image, it can be treated as a constant and (7) simplifies to $\mathcal{O}(\log_2 \gamma_r)$ which means the complexity of Step 2 and Step 3 combined becomes

$$\sum_{r=1}^{mn-1} \mathcal{O}(\log_2 \gamma_r) = \mathcal{O}(mn \log_2 mn). \tag{9}$$

Again, this is the total complexity of the algorithm since the initialisation is no worse.

Empirical evidence of the computational complexity of Algorithm 3 is shown in Figure 1. These results are indicative only since computation times will vary depending on

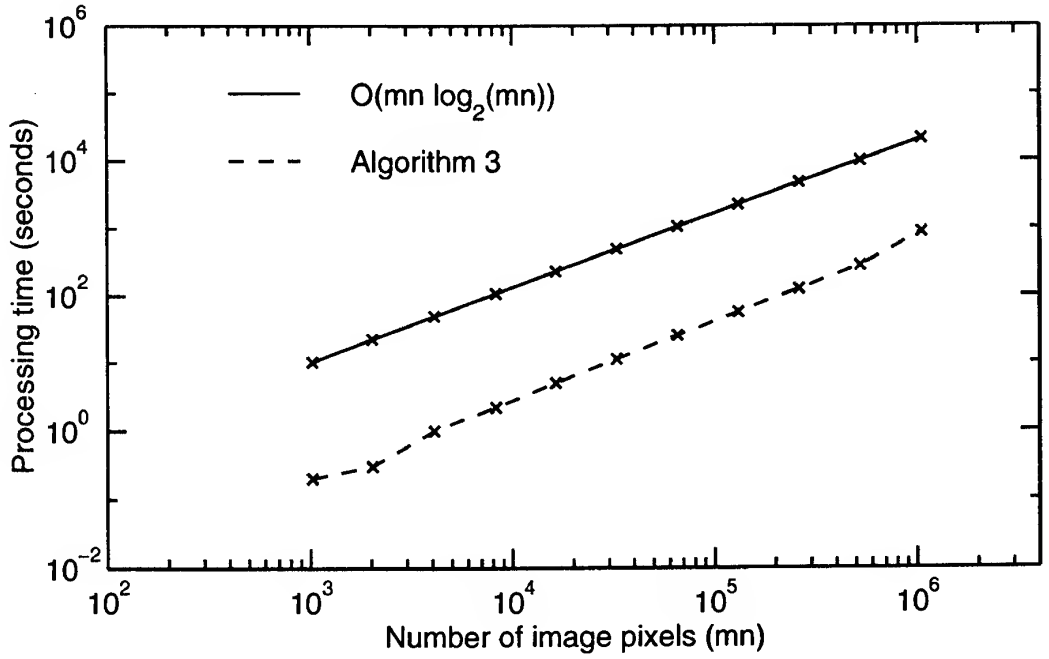


Figure 1: Typical computation times for Algorithm 3 on stripmap SAR images.

the nature of the imagery being segmented. The particular imagery used to produce the figure was provided by the Surveillance Systems Division (SSD) of DSTO. It was formed by a synthetic aperture radar (SAR) operating in stripmap mode and the area being imaged was farmland near Saddleworth, South Australia. The figure also shows a plot of $\mathcal{O}(mn \log_2 mn)$. We added this plot in order to get some insight into which of the complexity bounds, (8) or (9), is applicable. Note that, the figures axes are logarithmic and the plot of $\mathcal{O}(mn \log_2 mn)$ is effectively a straight line of slope 1. We would likewise expect a plot of $\mathcal{O}((mn)^2 \log_2 mn)$ to be effectively a straight line but with slope 2. Clearly the experimental data for Algorithm 3 lies along a straight line. While the slope of this line is larger than 1, it is still much closer to 1 than to 2. It follows (in this case at least) that complexity of Algorithm 3 is closer to (8) than (9).

3 Data Structures

Achieving a fast implementation of the full λ-schedule algorithm requires efficient data structures. We now present in detail the programming constructs for these data structures that are as efficient as we think is likely to be possible. The data structures and procedures reported here represent the culmination of analysing and rewriting three previous implementations [8, 11, 10], so we do not make our claim lightly.

3.1 Regions

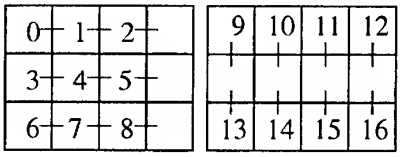
A region is 4-connected group of pixels designated as a single entity because it has a homogeneous characteristic. As the segmentation algorithm progresses, the regions will decrease in number and increase in size, *i.e.* the algorithm merges regions with the same character as it goes along. The algorithm is initiated on the trivial segmentation where every pixel is considered a separate region. Each region is identified by a unique region identifier, or *label*. Initially in the trivial segmentation of the image, this label corresponds to the pixel's offset within the original image when the image pixels are stored in row order. For example, a trivial segmentation of a 3×4 image would be labelled as shown in table 1.

		column			
		0	1	2	3
	0	0	1	2	3
row	1	4	5	6	7
	2	8	9	10	11

Table 1: Region labels for an example 3×4 image after the initial trivial segmentation of the image.

The single most important reason why the full λ-schedule algorithm can be made computationally efficient is that during the merging process it is not necessary to refer to the actual pixels that make up the region, just some summary statistics of each region. For each region, only its area (number of pixels), (array of) average value(s), and the labels of its neighbouring regions, are required. Note that an array of average values may be employed, rather than a single value to accommodate multi-band images or texture feature values at every pixel location. The neighbouring regions are recorded in a neighbouring pair (NP) list (section 3.3). The data structure used to implement a region within the full λ-schedule algorithm is expressed below in the C programming language.

```
typedef struct _tRegion
{
  double *gval;       /* array greyscale values for the layers in
                         the region */
  np_rbTree npList;   /* neighbouring pair list - a list of
                         labels to regions which have a common boundary
                         with the current region and the index of the
                         pairs made between the current region and these
                         neighbours */
  int    label;       /* unique region identifier */
  int    area;        /* area covered by region (in pixels) */
} tRegion;
```



(a) Horizontal (b) Vertical

Figure 2: Labelling of the (a) horizontal, and (b) vertical pairs for the trivial segmentation of the example 3×4 image.

3.2 Pairs

A pair is defined to be two neighbouring regions with a common boundary of vertical or horizontal components, *i.e.* the 4-connected boundary. Each pair is uniquely identified within the algorithm by the labels of its two instigating regions. Three parameters are also stored with each region pair: the common boundary length, the merge cost t (6), and the weighted intensity difference (the numerator of (6)). The common boundary length is the Hotelling distance along the boundary between the two regions in units of pixel width. The weighted intensity difference is stored in addition to the first two parameters even though it is simply the product of the two to save on computation, because all three are required. The structure used to represent a pair within the code is shown below.

```
typedef struct _tPair
{
  double t;              /* merging cost (t_ij) of the pair */
  double s;              /* value of weighted intensity difference which
                            remains constant under changes in the common
                            boundary length between the regions */
  int   r[2];            /* the indices to the paired regions in
                                    the region set */
  int   boundary_length; /* length of boundary between the paired
                                    regions */
} tPair;
```

The pairs are indexed during initialisation of the trivial segmentation; the horizontal pairs are indexed first, then the vertical pairs. The indices are incremented in the same left to right raster scan process used to label the regions. Using the example 3×4 test image in table 1, the resulting pair indices in the trivial segmentation for both horizontal and vertical pairings is shown in figure 2.

3.3 Neighbouring Pair List

When a pair of regions are merged to form a single region, the pair set values for any other pair which included either of the two merged regions will change. The neighbouring

pair (NP) list provides a fast and efficient mechanism to identify which pairs are effected by the changes.

Each region has an associated NP list. These lists contain the labels of all the regions that are contiguous with a particular region, and the index of the pair formed between them. Hence, stored within each node of an NP list is the label of the neighbouring region `rInd`, and the pair index `pInd`, as shown below.

```
typedef struct _np_rbNode
{
  int                rInd;    /* neighbouring region label */
  int                pInd;    /* index of pair formed between the current
                                 and neighbouring region */
  struct _np_rbNode *left;    /* pointer to left node (less than) */
  struct _np_rbNode *right;   /* pointer to right node (greater than) */
  struct _np_rbNode *up;      /* pointer to parent node (up) */
  unsigned char      red;     /* current node colour */
} np_rbNode;
```

The other parameters in the structure are used to implement the list efficiently as a red-black tree (Section 3.8).

After two regions are merged, the NP lists of the two parent regions are used to efficiently update pairs formed with neighbouring regions that are effected by the change. The NP lists of the two merged regions must also be merged to form the NP list of the newly formed region. Multiple entries due to a region being a common neighbour to both parent regions must be eliminated from the NP list of the new region. The elements of the NP lists are maintained in order of neighbouring region label to assist in eliminating duplicates.

As an example, consider the 3×4 image from section 3.1, table 1. Assuming the contents of each node in the NP list can be written in the form

$$\text{region\_label(pair\_index)}$$

the NP list for region 6 would be

$$\text{NP\_list}_6 \equiv 2(11) \rightarrow 5(4) \rightarrow 7(5) \rightarrow 10(15).$$

3.4 Region Set

The region set is an array of region records. The length of the array is not updated dynamically, but remains the same throughout the execution of the program. Its length is determined by the number of regions in the trivial segmentation of the image. Because each pixel is defined as a region in the trivial segmentation, the number of regions and hence the length of the region set will be $m \times n$ for an image of m rows and n columns. As regions are merged, entries within the region set become invalid because the larger region

label out of any pair of merged regions no longer refers to an extant region. Consequently, the data recorded in the invalid entry will not be further accessed by the algorithm.

For example, assume that regions p and q merge to form region pq. All the relative data associated with region pq is updated and recorded in region p's entry within the region set. The new region pq will be know as region p within the algorithm from this point. However, region q's entry will no longer be referenced during the algorithm, and the entry will not be used. Hence as the algorithm progresses, more entries within the region set will become invalid. This method of processing was chosen because the cost of updating dynamic data structures is very high computationally for little significant benefit in terms of memory usage.

For the initial trivial segmentation of an image, each entry in the region set represents an individual pixel within the image. The entries each have a label which corresponds to the offset of the pixel within the image when stored in row order, an area of 1 (pixel), and a greyscale array corresponding to the pixel's value in each layer of the image being processed. The NP list is computed for each region during the initialisation of the region set.

Expanding on our example 3×4 image in table 1, we shall assume the image has two greyscale layers. The resultant region set after the initialisation using the trivial segmentation of the two layer image can be represented diagramatically in figure 3, where the greyscale values for each layers are also shown.

3.5 Pair Set

As with the region set (section 3.4), the pair set is a static length array of pair records, where the initial length of the array is determined by the number of pairs in the initial trivial segmentation of the image in question. For an image of m rows and n columns, the initial number of pairs, and hence the length of the pair set, is $2mn - m - n$. This assumes that regions can only be paired through horizontal and vertical boundaries as stated in section 3.2. As the maximum number of pairs is known, a static implementation can be used without the risk of data overflows.

Each pair existing in the current segmentation K_r, is identified by a unique *pair index*, which is used to retrieve the relevant pair details from the pair set. For example, if the NP list of region p contains a node with the region label `nR` and pair index `nP`, then entry `nP` in the pair set array will contain the pair data for this pair.

The pair set array is static because as pairs are merged, the contents of the pair entry are not removed. Instead, as seen with the region set, they are ignored for the remainder of the algorithm, because the pair no longer exists in current and succeeding states of the segmentation.

3.6 Merge Candidate List

Before a merge can occur, the pair with the smallest merge cost t must be identified. These values are stored within the pair records of the pair set, but the pair set is not

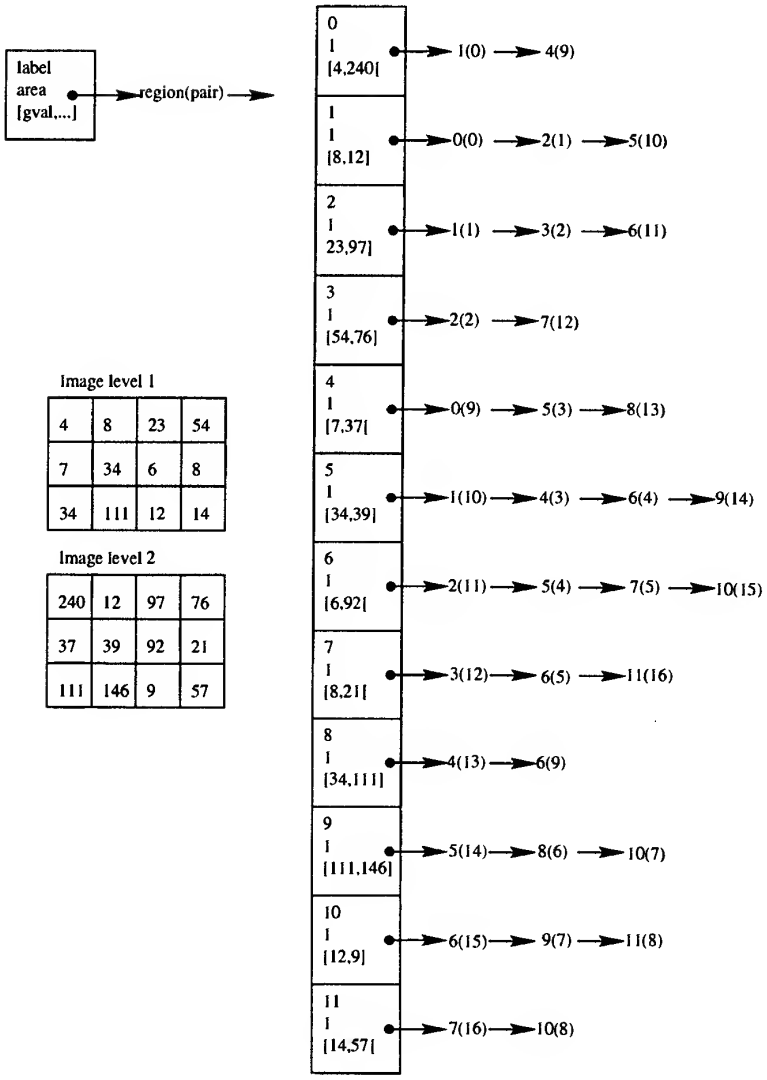


Figure 3: Structural representation of the region set for the example 3×4 2 greyscale layer image in table 1 after the trivial segmentation.

sorted in any order. Consequently, to identify the pair to merge in simple terms, we must search the entire pair set to

1. Identify pairs which still exist in segmentation K_r, and
2. From these existing pairs, identify the pair with the lowest $t_{i,j}$ value.

As the pair set is not dynamic, it make little sense to sort the values stored within the array, especially when we only require the merging suitability parameter, $t_{i,j}$, to be sorted. Consequently we create a list of indices into the pair set that are ordered by $t_{i,j}$ value called the *merge candidate list* to implement the two steps above.

Sorting the pairs is performed using the t values for each region pair as the primary key, with identical values then being sorted using the pair index as the secondary key. Each node in the merge candidate list is of the form shown below, where `t` and `index` are the programming constructs for t value and pair index respectively. Note that the other elements in the node are used to implement the red-black tree data structure (Section 3.8).

```
typedef struct rbNode_
{
  double          t;       /* merge cost of pair */
  int             index;   /* index pair entry within pair set */
  struct rbNode_ *left;    /* pointer to the left node (less than) */
  struct rbNode_ *right;   /* pointer to the right node (greater than) */
  struct rbNode_ *up;      /* pointer to the parent node (up) */
  unsigned char   red;     /* colour of link to the current node */
} e_rbNode;
```

The merge candidate list updates its contents dynamically during the course of the algorithm. As a pair is merged, that pair is removed from the merge candidate list. Similarly, pairs formed from neighbours of the merged regions are also removed. A new entry is created for the merged region to ensure the data in the list is in its correct order.

3.7 Merge List

The merge list records the indices of the merged regions and their associated merge cost t, and the greyscale values of each level of the newly formed region. This record gives the ability to recreate the image at any segmentation K_r without having to re-run the entire segmentation algorithm. A large proportion of the processing time of the segmentation algorithm is required to maintain the merge candidate list and the neighbouring pair lists of the remaining regions. By eliminating the maintenance on these data structures, the resultant image can be quickly recreated. The list is called the *merge list.* The data structure used in the code body to record these values is shown below.

```
typedef struct _tMergeNode
{
  double  lambda;  /* minimum merging cost for current iteration */
```

```
  double *gval;    /* array greyscale values for the image layers
                      in the newly formed region */
  int     plabel;  /* label of first region in the merged pair */
  int     qlabel;  /* label of second region in the merged pair */
} tMergeNode;
```

The merging process is performed from the trivial segmentation of $m \times n$ regions to the final segmentation of at least one remaining region. Hence $(2mn - m - n)$ merges are performed at most over the duration of the program. As the total length of the list is known prior to the execution of the algorithm, we can simply implement the merge list as an array of merge nodes defined in the listing above.

3.8 Balanced Trees

The complexity of searching, inserting and deleting elements in a binary tree is $\mathcal{O}(h)$, where h is the effective height of the binary tree in question. For binary trees built through random data sets, the height of the resultant tree is $\mathcal{O}(\log_2 N)$, where N is the number of entries within the tree. (With random data, entries are evenly divided or *balanced* on either side of any branching point. A balanced tree is defined to be one in which the height of left subtree at every node never differs by more than ± 1 from the height of its right subtree [3], p. 459.) Hence the computational complexity of operations on a binary tree is $\mathcal{O}(\log_2 N)$. However, if the data is not random, the worst case complexity of $\mathcal{O}(N)$ can occur, and the tree structure is termed *unbalanced*. Careful performance testing revealed that this was indeed occurring in a previous (suboptimal) implementation of the full λ-schedule segmentation algorithm [11]. Consequently unbalanced trees were one important reason why this previous implementation did not achieve the theoretical performance expected.

Red-black trees are a form of binary search tree that constrain the way the tree is constructed so that the tree is approximately balanced [3, 2]. There are many other alternatives, but red-black trees seem to be one of the more common choices, with comprehensive descriptions available [2]. An in-depth study into their efficiency for sorting, insertion and deletion indicated that they performed satisfactorily for our purposes. Consequently, in an attempt to achieve the $\mathcal{O}(mn \log_2 mn)$ complexity of sorting the list of pairs, the binary tree structures used in the previous implementation were replaced by *red-black* trees.

A red-black tree employs an extra bit of storage per node called its *colour*, which is either red or black. During insertion and deletion, constraints are placed on the way in which the nodes can be coloured on any path from the root to a leaf, to ensure that no path is more than twice as long as any other. Consequently, the resulting tree is approximately balanced.

A red-black tree has the following properties [2]:

- Every node is either red or black.
- Every leaf (end node) is black.

- If a node is red, then both its children are black.
- Every simple path from a node to a descendant leaf contain the same number of black nodes.

Due to the tree's approximate balanced nature, the complexity of the search, minimum, addition and remove operations are $\mathcal{O}(\log_2 N)$. This complexity meets the desired computation goal for the sorting process required by the full λ-schedule algorithm.

4 Procedures and Functions

Using the data structures described in the previous section, the steps involved in each merge of the full λ-schedule algorithm fall into the following four stages.

1. Identify the pair with minimum t to merge during the current iteration.
2. Update the merged region's parameters.
3. Update the neighbouring pair lists for the new region and its neighbours.
4. Update the merge candidate list.

This four stage process is guided by the manner in which the data structures interact in the algorithm (figure 4) and follows the concise description in Algorithm 3. Let us now consider these steps in more explanatory detail to show how the steps of Algorithm 3 relate to the programming constructs of the previous section. In the following sections, the implementation of the four steps outlined above shall be examined.

4.1 Identify Merger Pair

The first step in each successive merge of the full λ-schedule algorithm is to identifying the pair to be merged, *i.e.* the pair with the minimum t. When there is more than one of these, we take the pair with the minimum pair index. This is performed using the pair set and merge candidate list. The implementation only allows one merge candidate list to exist, with the implementation details hidden from the user. Only the ability to create and destroy the structure in memory, and add, remove and search for entries within the set are available to the user. The declaration file, `e_rbTree.h`, lists the external function available to the user. In comparison, the Pair Set is available globally in the segmentation code body, and represented through the data structure `A`, which is declared in the code body as

```
tPair *A;  /* pair set */
```

All future references to `A` refer to this programming construct for the Pair Set.

As mentioned, the algorithm starts out to form the segmentation at iteration r, denoted K_r, by locating the pair with the minimum t. The pair index of this pair is determined

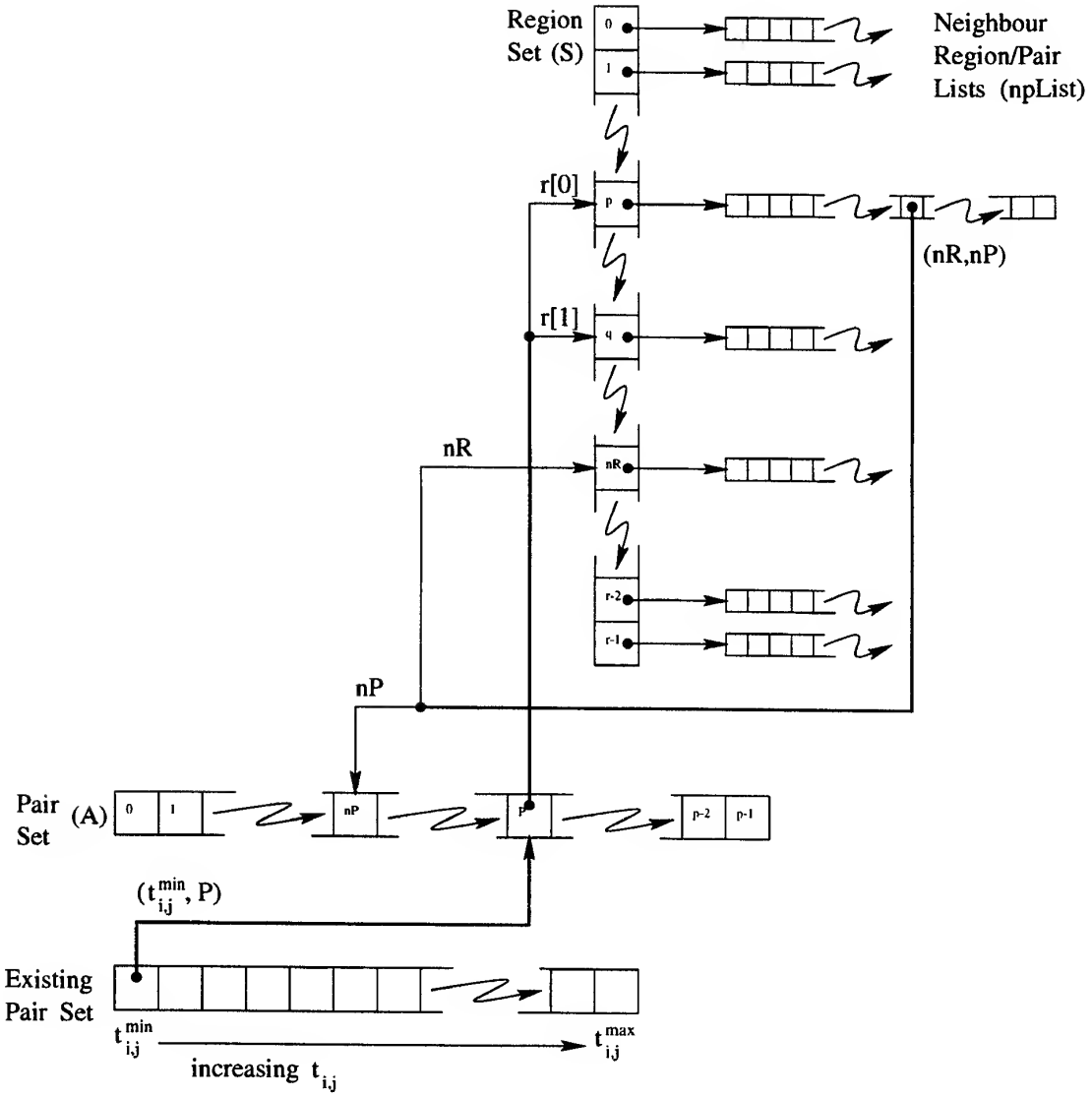


Figure 4: Interaction of the data structures used in the full λ-schedule algorithm for segmentation for the merger of two regions, p and q which correspond to the pair, P within an image.

by the merge candidate list search routine `e_minRbNode()`. Let the pair's pair index be denoted by `P`. The parameter denoting the pair's suitability for merging, t is also obtained. Using this index we can obtain the pair's full details within the Pair Set, `A`, by indexing into element `P` within the array. Hence the first step proceeds as follows:

- Identify the pair index of the pair with the minimum t using the merge candidate list.
- Obtain the pair by indexing into the pair set using `A[P]`.

4.2 Merge Regions

In the previous step we identified the pair to be merged whose index was `P`, and its associated entry in the pair set, `A[P]`. The next step involves merging the regions and updating the associated region data but does not include updating the NP lists. Within the code body, the region set is identified by the simple array, `S`. This data structure is globally available to the segmentation algorithm, and is declared as:

```
tRegion *S;  /* region set */
```

The first step of the region merging process is to identify the regions which create the pair, `A[P]`. In section 3.2, it was noted that the labels of the region creating the pair were recorded in the two dimensional array `r[]`, where the first entry will be referred to as region p, and the second entry referred to as region q. Hence, the labels of region p and q are `A[P].r[0]`, and `A[P].r[1]` respectively. The region labels are equivalent to the region's entry within the region set array, making the data structures for regions p and q equivalent to `S[A[P].r[0]]` and `S[A[P].r[1]]`, respectively. By convention, region p will always have the smaller label of the two regions being merged. This fact is enforced in the algorithm so that `S[A[P].r[0]].label` $<$ `S[A[P].r[1]].label` is always correct.

With regions p and q identified, the new region, pq, formed from merging these two regions, can now be constructed. The data associated with region pq replaces region p within the region set. The area and the greyscale values for each layer within the image are calculated using

$$a_{pq} = a_p + a_q, \tag{10}$$

$$u[n]_{pq} = \frac{a_p u[n]_p + a_q u[n]_q}{a_{pq}}, \tag{11}$$

respectively, where a represent the region area, and $u[n]$ represents the n-th layer greyscale value for the region. The results are written to the region data structure which formerly stored the information for region p.

To assist the process of updating the NP list of the merged pair, the reference to regions q and p are removed from the NP lists of regions p and q respectively. As these regions have been merged, they no longer technically exist, and this also removes the requirement to test each neighbour region to see if it one of the pair being merged.

The labels of regions p and q and the resultant greyscale layers are recorded within the merge list entry r to allow the image of segmentation K_r to be recreated.

4.3 Update Neighbouring Pair Lists

The t values for the pairs made between region p or q and their neighbouring regions must be updated to correspond to the new region, pq. This is a two step process. First the pairing must be identified and its old t value removed, and secondly, the new t calculated for the new pairing and this value then recorded and sorted. The NP list is designed to facilitate this process.

Each region record within the region set `S`, has an associated NP list, denoted by the variable `npList`. The pairs created by the regions p or q and their associated neighbours recorded in their respective NP lists will be effected when the new region pq is created. The union of these two lists (without duplicates) contains all the region pairs effected by merging p and q. As region pq replaces region p in the region set, the union of the two lists is simply achieved by merging the NP list of region q (`S[A[P].r[1]].npList`) with that of region p (`S[A[P].r[0]].npList`). This is undertaken by the following steps.

Updating the neighbour's region pairs is performed on a record by record basis by traversing the NP list of the particular region. Let `nP` and `nR` denote the pair index and region label, respectively, of the current node in the NP list corresponding to the neighbouring region r. Then looping through the values of r, the update process proceeds as follows.

- Remove the pair (p, r) entry from the merge candidate list, using the old t value (`A[nP].t`) and pair index (`nP`) as the search keys.
- Obtain the neighbour data from the region set entry `S[nR]` and determine the new $s_{pq,r}$ value for the pair (pq, r) given by

$$s_{pq,r} = \frac{a_{pq}a_r}{a_{pq} + a_r} \sum_{i=1}^{n} (u[i]_{pq} - u[i]_r)^2 \tag{12}$$

  where n is the number of layers. Note that this is independent of the common boundary length of the pair (pq, r).
- Update the $s_{pq,r}$ value for the pair (pq, r) in the pair set.

Entries for the neighbours of region q must be merged into the existing list of neighbours for the region pq. The process involved in merging region q's neighbours into pq's NP list can be summarised as follows. Note that the label p is reused to refer to the merged region pq.

- Remove the pair (q, r) entry from the merge candidate list using the old $t_{q,r}$ value (`A[nP].t`) and pair index (`nP`) as the search keys.
- Obtain the neighbour data from the region set entry `S[nR]` and attempt to merge this entry into pq's NP list.
- SUCCESS: the neighbouring region is a new entry in region pq's NP list.

  - Update the region labels for the pair (pq, r), changing the reference to q to be that for the combined region (which by convention is the new region p). Also, swap the region indices p and r in the pair if necessary to ensure that the constraint `S[A[nP].r[0]].label < S[A[nP].r[1]].label` is always satisfied.
  - Recalculate the component of t independent of the common boundary length for the pair using (12).
  - Remove the entry referencing region q as a neighbour from the neighbouring region's NP list. Reinsert a new entry within this list referencing region p as a neighbour. Nodes are removed using the NP list manipulation routine `np_removeRbNode()`, and added using `np_addRbNode()`.

- FAILURE: the neighbour region already exists within pq's NP list. Hence it was a neighbour to both original regions p and q.

  - Update the common boundary length between the neighbouring region and region pq to include the common boundary length contribution from the merged region q.
  - Remove references to region q from the neighbours NP list, as this region no longer exists.

Due to the difference in functionality dependent on which NP list is being processed, two separate functions are used, each clearly identified to which list is operates on, p or q.

4.4 Update Merge Candidate List

The last step in each merge operation is to update the t values in each entry of pq's NP list and also update their values in the merge candidate list E. Each node within pq's NP list is visited, and using the contents of the node the following operations are performed.

- Calculate and update the value t for the pair using the calculated values of the pair common boundary length and the component of t independent of the common boundary length using

$$t_{p,q} = \frac{s_{p,q}}{l\left(\delta(O_p, O_q)\right)} \tag{13}$$

  where $l()$ refers to the length of a component, and $\delta(O_p, O_q)$ denotes the common boundary between the regions O_p and O_q.

- Update the pair's value of t in the merge candidate list.

Each entry added to the merge candidate list is performed using the node addition routine `e_addRbNode()` which inserts and sorts the new entry. Once the traversal has completed, the two regions have successfully been merged, and if the exit criterion has not yet been reached, the process starts once again by selecting the next pair with the minimum t and pair index.

5 Enhancement to the Algorithm: Pruned Merge Candidate List

One of the more expensive computational steps of the full λ-schedule algorithm is sorting the merge candidate list $\mathcal{E}$ according to merge costs. The list can be very long (for a rectangular image of $m \times n$ pixels it is initially contains $\alpha_0 = 2mn - m - n$ elements) and it needs resorting each time a merge is performed. Moreover, the merges costs of all pairs which involve the merging regions are likely to change and so, in general, more than one entry will be out of place. In this section we report on an enhancement to the full λ-schedule algorithm which helps overcome this computational burden. The enhancement utilises the concept of *locally best merges* (to be defined in the next section) and involves pruning $\mathcal{E}$ so that it only contains the locally best merges. The effect of this change is to significantly reduce the size of $\mathcal{E}$ and its implementation has resulted in an order of magnitude decrease in the algorithm computation time.

5.1 Locally best merges

The idea underlying the enhancement is that, since a region merge operation only affects the merge costs in a local area of the segmentation, it makes sense to sort the affected merge costs first and then only enter the best (cheapest) ones into the merge candidate list. While this idea is natural it requires some care in defining what is meant by a locally best merge. We need to ensure that the set of locally best merges is easily updated and that the globally best merge is included. To do this, we first need to be more precise about what is meant by best and globally best.

Given a set of potential region merges in a segmentation, we define the *best merge* to be the one with the smallest merge cost. This then allows us to define the *globally best merge* for a segmentation to be the best merge in the set of all possible merges. However, in order to avoid unnecessary complications in our algorithm, we need resolve the ambiguity in the case of tied merge costs. We do so by choosing the *best merge* in the case of tied costs to be the one with the smallest index in the pair set $\mathcal{A}$. While this means of resolving tied merge costs is somewhat arbitrary and there are other alternatives, we believe that in practice the choice of definition will not affect the resulting segmentations. Our reasoning is that the order of merging for tied merge costs will only be significant if the costs are also globally the best and in that case, the merge boundaries represent the least significant image structures.

We can now give our main definition. We say that a pair of neighbouring regions (O_i, O_j) represents a *locally best merge* if it is the best merge in the set of all merges which involve either O_i or O_j (or both). As stated above, our enhancement to the full λ-schedule algorithm is to prune the merge candidate list $\mathcal{E}$ so that it only contains the locally best merges. It should be evident from the definitions that globally best merge is also a locally best merge and so our pruning of the list will never remove the first element $\mathcal{E}_0 = (t_0, P_0)$. It follows that, exactly the same segmentations will be produced by the enhanced algorithm.

As an indication of the savings that pruning the merge candidate list leads to, observe

that at any stage in the segmentation process the number of locally best merges is at most half the number of regions (since each locally best merge accounts for two regions), where as, the total number of possible merges is greater than the number of regions less one (since each merge reduces the number of regions by one and merging can continue until only one region is left). Thus by using locally best merges only, we are guaranteed of halving the size of the $\mathcal{E}$ list. Experiments have shown that in general the savings are much greater.

It only remains to show that all locally best merges can be easily found. Returning to the definition, it is evident that if (O_i, O_j) is a locally best merge then O_j is the best neighbour for O_i to merge with and *visa-versa*. It follows that we can find all the locally best merges by first scanning the list of regions and for each one determining its best merging neighbour. By reviewing a list of best merging neighbours and looking for instances where the best merging neighbour of a region says likewise that the original region is its best merge, we can find all the locally best merges. Note that, while this provides the thinking behind our implementation, we make the search much more efficient by using our linked data structures, as described in the next section.

5.2 Description of the algorithm

In this section we show how the scheme just described for finding all locally best merges can be implemented efficiently and then go on to describe full details of the enhanced algorithm.

Our implementation requires extending the previous data structures by adding an extra variable to each entry of the region list $\mathcal{S}$ and two new flags to each entry of the pair list $\mathcal{A}$. Thus the typical entries of these lists now have the form

$$\mathcal{S}_i = (I_i, a_i, \mathbf{u}_i, \rightarrow_{\mathcal{N}_i}, bestp(i)) \tag{14}$$

and

$$\mathcal{A}_i = (p, q, s_{p,q}, t_{p,q}, b_{p,q}, bflag(p,q), eflag(p,q)). \tag{15}$$

The new item, $bestp(i)$, in (14) records the index in the pair list $\mathcal{A}$ of the best merge for the region O_i. The value of this index is determined by using the NP list $\mathcal{N}_i$ to search the merge cost information in in the relevant entries of $\mathcal{A}$ for O_i's best merging neighbour. Note that, since each entry $\mathcal{A}$ also records the indexes of both the regions involved, we can obtain the region set index of O_i's best merge from its *bestp* variable at any time we want.

The item $eflag(p,q)$ in (15) records whether or not the corresponding region pair is a locally best merge and hence is used to maintain the pruned merge candidate list $\mathcal{E}$. As mentioned, our method of setting this flag is to check whether or not both regions concerned say the other is its best merging neighbour. To help perform this check, we use the other new flag $bflag(p,q)$. This second flag is set when at least one of the associated pair of regions says the other is its best merge and is cleared otherwise. Note that if the *elfag* is set then the *bflag* must be too but the converse need not be true; the *bflag* can be set while the *eflag* is cleared. In this later case, only one region is saying the other is its

best merge. Note also that in this latter case we do not know which region is saying the other is its best merge, that information is only available indirectly via the *bestp* variables.

It only remains to describe how the enhanced algorithm initialises the quantities *bestp*, *bflag* and *elfag* and then updates them after each merge operation. Of course, this initialisation and the updates are in addition to those of the full λ-schedule as described in Algorithm 3. Fortunately, the new steps are essentially independent of the old ones and form two separate new steps of the algorithm which we present below as Step (1a.) and Step (2a.). Again, as well as describing each step, we have recorded its computational complexity (unless it is $\mathcal{O}(1)$) and as before, the variables γ_r, $\beta_{p,r}$, etc. are fixed at their maximum values for the current step. We will discuss the complexity of the algorithm as a whole in the next section. Note that as with the full lambda schedule algorithm we make good use of the linking in our data structures to obtain speed and efficiency. Also a final word on notation. Recall that in Algorithm 3 we used the notation

$$t_{P_i} = t_{p,q}$$

where the indices satisfy (15) above. We will likewise use the notation

$$bflag_{P_i} = bflag(p,q) \qquad \text{and} \qquad eflag_{P_i} = eflag(p,q)$$

in the following.

Algorithm 4

1a. **Initialise the merge candidate list.** (Assume Step (1) of Algorithm 3 has just completed but with Step (1.(c)) deleted.)

Initialise the quantities *bestp*, *bflag* and *elfag* and adjust the merge candidate list $\mathcal{E}$ accordingly as follows:

(a) For each region O_i in the region list $\mathcal{S}$ do: $\mathcal{O}(mn) \times \ldots$

i. Search the costs t_{P_j} of merging O_i with its neighbours $\mathcal{N}_{i,j} = (j, P_j)$ for its best merge and fix P_j to be the corresponding index. $\mathcal{O}(1)$

ii. Set $bestp(i) = P_j$.

iii. **If** $bflag_{P_j} = 0$ **then** set $bflag_{P_j} = 1$
Else set $eflag_{P_j} = 1$ and insert (t_{P_j}, P_j) into $\mathcal{E}$. $\mathcal{O}(\log_2 \gamma_0)$

2a. **Update the merge candidate list.** (Assume Step (2) of Algorithm 3 has just completed but with Step (2.(h)ii), Step (2.(i)ii) and Step (2.(j)ii) deleted and assume that p and q are as described in Step (2.(c)).)

Update the quantities *bestp*, *bflag* and *elfag* and adjust the merge candidate list $\mathcal{E}$ accordingly as follows:

(a) Use $\mathcal{N}_p$ to loop through the neighbours O_n of O_p and update *bestp*, *bflag* and *elfag* for each O_n as follows: $\mathcal{O}(\beta_{p,r}) \times \ldots$

i. Obtain the pair set index P_n for the pair (O_n, O_p) from $\mathcal{N}_{p,n} = (n, P_n)$.

ii. Use $\mathcal{S}_n$ to locate $bestp(n)$ and set $P_i = bestp(n)$.

iii. Use $\mathcal{A}_{P_i}$ to obtain the region index, say k, of O_n's previous best merge

iv. **If** $k = p$ **or** $k = q$ **then**
(O_n's previous best merging neighbour was either O_p or O_q so completely recalculate its best merge)

A. Set $bflag_{P_i} = 0$.

B. Search the costs t_{P_j} of merging O_n with its neighbours $\mathcal{N}_{n,j} = (j, P_j)$ for its best merge and fix P_j to be the corresponding index. $\mathcal{O}(\beta_{n,r})$

C. Set $bestp(n) = P_j$.

D. **If** $bflag_{P_j} = 0$ **then** set $bflag_{P_j} = 1$
Else set $eflag_{P_j} = 1$ and insert (t_{P_j}, P_j) into $\mathcal{E}$. $\mathcal{O}(\log_2 \gamma_r)$

Else
(O_n's previous best merging neighbour was neither O_p nor O_q)

If $t_{P_n} < t_{P_i}$ **or** $t_{P_n} = t_{P_i}$ and $P_n < P_i$ **then**
(Merging O_p with O_n is better than O_n's previous best merge)

A. **If** $eflag_{P_i} = 1$ **then** set $eflag_{P_i} = 0$ and delete (t_{P_i}, P_i) from $\mathcal{E}$. $\mathcal{O}(\log_2 \gamma_r)$

Else set $bflag_{P_i} = 0$.

B. Set $bestp(n) = P_n$ and $bflag_{P_n} = 1$.

(b) Update *bestp*, *bflag* and *elfag* for region p as follows:

i. Search the costs t_{P_m} of merging O_p with its neighbours $\mathcal{N}_{p,m} = (m, P_m)$ for its best merge and fix P_m to be the corresponding index. $\mathcal{O}(\beta_{p,r})$

ii. Set $bestp(p) = P_m$.

iii. **If** $bflag_{P_m} = 0$ **then** set $bflag_{P_m} = 1$
Else set $eflag_{P_m} = 1$ and insert (t_{P_m}, P_m) into $\mathcal{E}$. $\mathcal{O}(\log_2 \gamma_r)$

5.3 Computational complexity

In this section we update the computational complexity calculations of Section 2.3.2 to take account of the changes introduced in Algorithm 4. We begin with the initialisation step. By combining the complexities of the sub-steps as listed in Algorithm 4 it is clear that the complexity of Step 1a is $\mathcal{O}(mnlog_2\gamma_0)$. Using $\gamma_0 < L_0 = mn$ this simplifies to $\mathcal{O}(mn \log_2 mn)$. To get the complexity of the complete initialisation step we need to add the complexity of Step 1. However, the complexity of Step 1 in Algorithm 4 is no worse than that of Step 1 in Algorithm 3 and we know that was $\mathcal{O}(mn \log_2 mn)$. It follows that the complexity of the complete initialisation step is $\mathcal{O}(mn \log_2 mn)$.

Next we consider the iterative part of the algorithm. By combining the contributions of the sub-steps as listed in Algorithm 4 it is not hard to show that the computational complexity of Step 2a is

$$\mathcal{O}\left(\sum_{n=1}^{\beta_{p,r}}(\beta_{n,r} + \log_2 \gamma_r) + \beta_{p,r} + \log_2 \gamma_r\right) = \mathcal{O}\left(\sum_{n=1}^{\beta_{p,r}}(\beta_{n,r}) + \beta_{p,r} \log_2 \gamma_r\right). \quad (16)$$

Using $\beta_{\max}$, as defined in Section 2.3.2, we can simplify this to

$$\mathcal{O}(\beta_{\max}^2 + \beta_{\max} \log_2 \gamma_r). \tag{17}$$

However, the only bound on $\beta_{\max}$ we are sure of is $\beta_{\max} \leqslant L_r$ which leads to the estimate $\mathcal{O}(L_r^2 + L_r \log_2 \gamma_r)$ and unfortunately the term $\mathcal{O}(L_r^2)$ means this estimate is worse than the equivalent one for Step 2 in Algorithm 3. We need to be more careful in our calculation.

Our poor estimate arose from bounding the term $\mathcal{O}(\sum_{n=1}^{\beta_{p,r}} \beta_{n,r})$ in (16) too crudely. We can do better by analysing the corresponding steps of the algorithm in more detail. In other words, we want to derive a better estimate of the complexity of Step 2a.(a) when only the sub-step, Step 2a.(a)iv.B, is taken into account. The steps of algorithm involved are: visiting each of the neighbours O_n of O_p; and while at each O_n, obtaining the merge costs t_{P_j} of merging O_n with each of its neighbours. During this process, each merge cost t_{P_j} can only be visited at most twice since a merge cost only involves two regions. It follows that the complexity of this process can be estimated as $\mathcal{O}(2\alpha_r)$, where α_r is the current number of region pairs. Inserting this estimate into (16) shows that complexity of Step 2a is $\mathcal{O}(\alpha_r + \beta_{p,r} \log_2 \gamma_r)$. We further simplify this by using $\beta_{p,r} \leqslant L_r$ to get

$$\mathcal{O}(\alpha_r + L_r \log_2 \gamma_r)$$

which is a much better estimate than $\mathcal{O}(L_r^2 + L_r \log_2 \gamma_r)$.

To complete our calculation of the complexity of the iterative part of the algorithm we first sum our estimates of the complexity of Step 2a as the algorithm runs to completion. This gives

$$\sum_{r=1}^{mn-1} \mathcal{O}(\alpha_r + L_r \log_2 \gamma_r) = \mathcal{O}((mn)^2 \log_2 mn) \tag{18}$$

where we have used the facts that $\alpha_r < 2mn - m - n - r$ (since at least one pair of regions is removed at each step of the algorithm), $L_r = mn - r$ and $\gamma_r < L_r = mn - r$. It only remains to take account of Step 2. However, the complexity of Step 2 in Algorithm 4 must be less than that of Step 2 in Algorithm 3 and we know that was no worse than $\mathcal{O}((mn)^2 \log_2 mn)$. It follows that (18) is the complexity of the total iterative part of Algorithm 4. Since the initialisation step is less complex, (18) is in fact the complexity of the complete algorithm and we are done.

If on the other hand $\beta_{\max}$ is independent of the size of the image, it can be treated as a constant and (17) simplifies to $\mathcal{O}(\log_2 \gamma_r)$. In this case the complexity of Step 2 and Step 3 combined becomes

$$\sum_{r=1}^{mn-1} \mathcal{O}(\log_2 \gamma_r) = \mathcal{O}(mn \log_2 mn). \tag{19}$$

Again, it is not hard to see that this is in fact the complexity of the complete algorithm since no other part has greater complexity.

In Figure 5, we compare the computation times of Algorithm 3 and Algorithm 4. The calculations were produced using the same imagery as for Figure 1. Again we comment

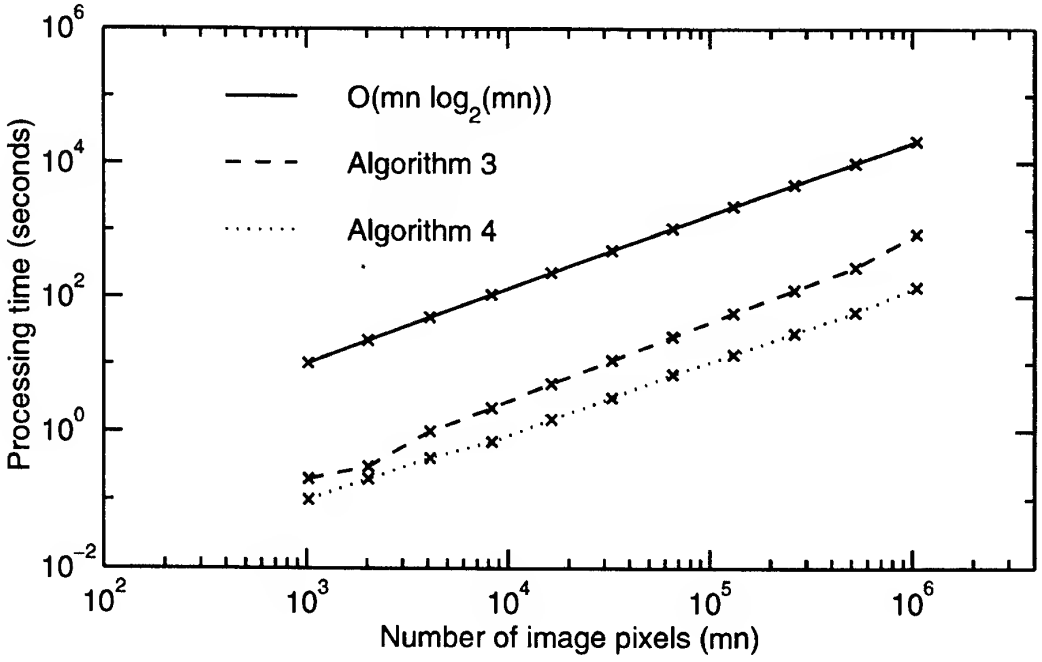


Figure 5: Typical computation times for Algorithm 3 and Algorithm 4.

that these results are indicative only since computation times will vary depending on the nature of the imagery being segmented. It can be seen from the figure that Algorithm 4 is roughly an order of magnitude faster than Algorithm 3. This is despite the fact that our estimates of their complexities are the same. The explanation is that Algorithm 4 has a much better constant of proportionality in its complexity than Algorithm 3. This is not surprising given the better bound we have on γ_r for Algorithm 4.

As before, we have also added a plot of $\mathcal{O}(mn \log_2 mn)$ in order to get some insight into which of the complexity bounds, (18) or (19), is applicable. It can be seen from the figure that the data for Algorithm 4 lies on a straight line which is parallel to the plot of $\mathcal{O}(mn \log_2 mn)$. Consequently, the empirical evidence is that Algorithm 4 has complexity $\mathcal{O}(mn \log_2 mn)$. This is not true for Algorithm 3 since the data appear lie on a line with slope greater than 1. Thus Algorithm 4 also appears to have better scaling properties than Algorithm 3. We conclude with comment that the best that could be expected of a segmentation algorithm is complexity $\mathcal{O}(mn)$. Since the effect of the term $\log_2 mn)$ is only minor, it follows that Algorithm 4 is doing about as well as we could expect.

5.4 Data structures and procedures

In this section we outline the implementation of Algorithm 4. We begin with the data structures. It is easily seen by checking the variables used in Algorithm 4 that the only new quantities needed are *bestp*, *bflag* and *eflag*. Further, the best way of incorporating them into our existing data structures is indicated in equations (14) and (15). Thus we

implement *bestp* by adding

```
int     bestp;    /* pair set index of the best merge neighbour */
```

to the declaration of the type `_tRegion` and we implement *bflag* and *eflag* by adding

```
int     bflag;    /* this flag is set if and only if at least one region
                     of the pair is saying the other is its best merge */
int     eflag;    /* this flag is set if and only if both regions
                     of the pair are saying the other is its best merge */
```

to the declaration of the type `_tPair`.

The implementation of Algorithm 4 also requires some new procedures. First, we have implemented all of Step (1a.) in a procedure which we call `init_EList()`. This procedure is a direct mirror of the sub-steps in Step (1a.) and needs no further explanation. Next, we have implemented all of Step (2a.) in a procedure which we call `updateEList()`. Again, the code in `updateEList()` essentially mirrors the description of Step (2a.) and we do not need to go into the details with one exception. The exception is that we avoid duplication of code for the searches at Step (2a.(a)ivB) and Step (2a.(b)i) by introducing a second procedure, called `updateBestP()`. This second procedure is very simple. It merely compares the merge cost of the current merging neighbour with the best so far and updates the pair set index of the best merge if necessary.

6 Supporting Procedures and Functions

6.1 Recording Segmentation Data

As it is currently implemented, the full λ-schedule algorithm terminates when all the pixels have finally been merged into a single region. This is not a particularly useful result. However, as previously mentioned, data has been recorded about each merge operation that allows the segmentation at any (useful) step to be recreated. In this section we examine the three data structures recorded: a merge list, a segmentation mask, and segmented image layers.

The merge list on its own contains all the information required to construct the segmentation at any stage of the algorithm. However, for large images the full merge list can be very large and so we usually truncate the list to only allow the construction of segmentations during the later stages of processing because they are the ones of principal interest. For example, the merge list for a single layer $2,048 \times 2,048$ image would occupy a file of approximately 134MB in size. If we are only interested in the segmentations corresponding to the last 40,000 merges, the merge list data required could be stored in a 1.28MB merge file. Truncating the merge list in this fashion requires that a segmentation mask and segmented image layers also be recorded for the starting point of the truncated merge list.

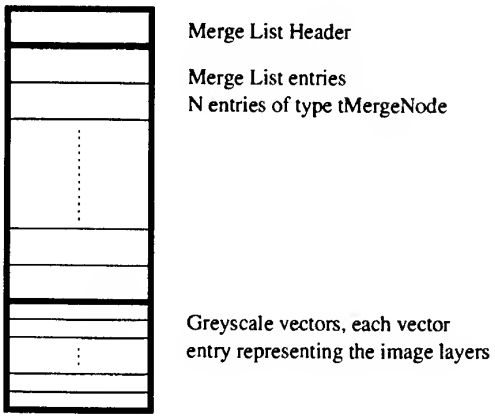


Figure 6: An overview of the structuring of the merge list data file.

Segmentation Mask

The segmentation mask codes the regions of a segmentation K_r at iteration r using a single distinct colour for each. The mask is stored as a portable pixmap (PPM) image, allowing just over 16 million (2^{24}) different regions to be represented. (Consequently, this imposes a limit of $4,096 \times 4,096$ to the size of a square image that can be correctly reconstructed in the current implementation.) This is achieved using the 24 least significant bits associated with the region label, allocating 8 bits to each of the red, green and blue components of the label.

Greyscale Layer Images

Each region of a segmentation K_r has an associated greyscale vector, one element for each layer of the image. The greyscale values represent the mean value of their respective layers across a particular region. In the greyscale layer images each pixel is assigned the corresponding region's value for each layer, storing them as portable grey map (PGM) image files.

Merge Files

The merge list records all the region merges performed in the execution of the segmentation algorithm after a designated starting point. The structure of the data within the merge list file is as follows (figure 6).

The merge file header specifies the starting segmentation from which the list was recorded as well as the number of merges recorded and the number of layers. The second block of the file contains the merge entries, corresponding to the data structure `tMergeNode`. The third block of the file contains the greyscale vectors associated with each merge entry. The greyscale vectors are recorded separately to simplify reading and writing the data because their length is not known at compile time.

6.2 Image Recreation

In the previous section, we presented the data structures that must be saved to allow the recreation of an arbitrary segmentation. In this section we present how to recreate a segmentation from this data.

6.2.1 Image Recreation Data Structures

The *pixel list* and *region list* are the two principal data structures used to reconstruct the image.

Pixel List

The pixel list is a linked list data structure, where each node contains the (row order) index of a pixel:

```
typedef struct _tPixelNode
{
  struct _tPixelNode *next;   /* next element in the list */
  int                 index;  /* pixel index within image */
} tPixelNode;
```

Each list is anchored through a structure which records the first (head) and final (tail) nodes as shown below.

```
typedef struct _tPixelList
{
  tPixelNode *head;  /* head node of the list */
  tPixelNode *tail;  /* tail node of the list */
} tPixelList;
```

Region List

The region list performs much the same task as the region set in the segmentation algorithm, in that it records the data associated with each region in segmentation K_r. It has two components: a greyscale vector and a pixel list for each region:

```
typedef struct _tRegionNode
{
  double     *gval;       /* greyscale array of all pixels in region */
  tPixelList  pixelList;  /* list of pixels in current region */
} tRegionNode;
```

As with the region set, this structure is static in length, and its length is determined by the number of pixels in the image being segmented. As the algorithm progresses, the length of the list remains constant, but the number of entries that are valid reduces by one each time a merge indicated by the merge list is performed.

6.2.2 Reconstruction

Recreating the segmentation mask and segmented image for a particular segmentation K_r is a two step process. Firstly, the region list must be initialized to the segmentation at the start of the recorded merge list. Secondly, the merges indicated in the merge list are carried out in order until the desired segmentation is reached.

Initializing the region list starts with the recorded segmentation mask and greyscale image layers and processes each pixel as follows.

- Identify the region O to which the current pixel belongs.
- If O's pixel list is empty, copy the current pixel's greyscale vector into region O's greyscale vector.
- Add the current pixel to O's pixel list.

When this process has considered every pixel, the region list will be initialised to the segmentation at the start of the merge list, allowing any of the segmentations past this to be recreated using the merge list.

The process of reconstructing the desired segmentation proceeds by looping over the following steps.

- Does the current segmentation meet the exit conditions, specified by either iteration number or a t that must be exceeded?
- Merge the region pair (p, q) specified in the current element of the merge list. Append the pixel list of region q onto that of p's pixel list. Reset q's pixel list to a null set.
- Update the greyscale array of region p with the value recorded in the corresponding element of the merge list.

Hence when the exit criterion has been reached, a number of regions in the region list will have empty pixel lists (null sets), and these denote regions which no longer exists in the image for segmentation K_r. Regions with valid (non null) pixel lists are a component of the image for the segmentation desired.

As an example, assume the image for segmentation K_r for our example 3×4, two layer image with pixels indexes as shown in table 1, has a region, p. This region has a greyscale vector [225,17] and covers the pixels of indexes 1, 4, 5, 9 and 10. The entry for region p in the region list can structurally be represented as shown in figure 7. The generated two layer greyscale image would have the first layer pixels of index 1, 4, 5, 9 and 10 assigned to the value 225, while the same pixels in the second layer were assigned to the value of 17. The mask is created in a similar manner, but the greyscale value written to the image is set to the current region label, in this case p. The resultant greyscale image layers and segmentation mask can be seen in figure 8.

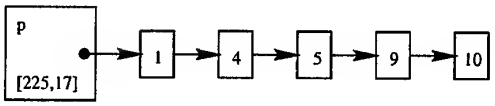
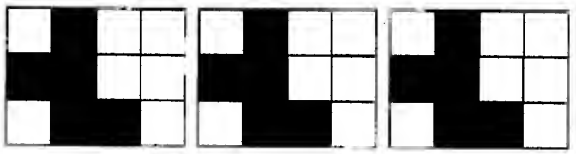


Figure 7: Region list entry p for the segmentation K_r. The region list entry records the greyscale vector for the region as well as the pixels associated with the region.



(a) Layer 1 (b) Layer 2 (c) Mask

Figure 8: First (a) and second (b) greyscale layers for region p in the segmented image and the corresponding segmentation mask (c) for segmentation K_r for the test 3×4 test image.

7 Conclusion

We have presented the details of a novel fast implementation of the full λ-schedule algorithm for segmentation. The algorithm trades increased memory usage to gain a low computational complexity of the merging process, and the result is an algorithm with speed on the order of that of the Fast Fourier Transform algorithm, the benchmark for fast algorithms. This algorithm will form the basis of a segmenting detection algorithm in the Analysts' Detection Support System.

Future work will involve testing the utility of the algorithm for target detection and extending the class of image models it uses. The initial steps in both these directions have already been taken. The algorithm has been incorporated into the ADSS software suite and once the coding of the entire suite has been completed and suitable data is available we will be able to test its target detection utility. On the second point, the algorithm described here only allows piecewise constant image models. This means some of the more subtle features of the SAR imagery cannot be exploited. However, progress has since been made on expanding the class of applicable image models to include piecewise polynomial models. It is anticipated that implementing these models will only involve a few local changes to the segmentation code and that these changes will not effect the computational complexity of the algorithm.

References

1. Abramowitz M., Stegun I. A., Handbook of Mathematical Functions, Dover, New York, 1968.

2. Cormen T. H., Leiserson C. E., Rivest R. L., Introduction to Algorithms, MIT Press, Cambridge, MA, 1992.

3. Knuth D. E., The Art of Computer Programming, Volume 3 Sorting and Searching, second edition, Addison Wesley Longman, 1998.

4. Koepfler G., Lopez C., Morel J. M., "A multiscale algorithm for image segmentation by variational methods", SIAM Journal of Numerical Analysis, 31(1), 1994, pp. 282–299.

5. Morel J-. M., Solimini S., "Variational Methods in Image Segmentation" Birkhäuser, Boston, 1995.

6. Mumford D., Shah J., "Boundary detection by minimizing functionals", Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, I, 1985, pp. 22–26.

7. Redding N. J., "Design for the Analysts' Detection Support System for Broad Area Aerial Surveillance", Defence Science and Technology Organisation, Australia, Technical Report DSTO-TR-0746, 1998.

8. Redding N. J., Crisp D. J., Tang D., Newsam G. N., "An efficient algorithm for Mumford-Shah segmentation and its application to SAR imagery", Digital Image Computing: Techniques and Applications, 1999.

9. Redding N. J., Robinson D. J., "Prescreening Algorithm Performance in the Analysts' Detection Support System", Defence Science and Technology Organisation, Australia, Technical Report DSTO-TR-0878, 1999.

10. Robinson D. J., "Unpublished notes on full λ-schedule segmentation algorithm performance", 1999.

11. Tang D., Redding N. J., Crisp D., Schroeder J., "Implementation of the Full λ Schedule Algorithm for Segmentation", Cooperative Research Centre for Sensor Signal and Information Processing, CSSIP Client Report 10/99, 1999.

DISTRIBUTION LIST

Implementation of a Fast Algorithm for Segmenting SAR Imagery
David J. Robinson, Nicholas J. Redding and David J. Crisp

	Number of Copies
DEFENCE ORGANISATION	
Task Sponsor	
DD JP129, DG Aerospace Division	1
S&T Program	
Chief Defence Scientist FAS Science Policy AS Science Corporate Management Director General Science Policy Development	1
Counsellor Defence Science, London	Doc Data Sht
Counsellor Defence Science, Washington	Doc Data Sht
Scientific Adviser to MRDC Thailand	Doc Data Sht
Scientific Adviser Joint	1
Navy Scientific Adviser	Doc Data Sht
Scientific Adviser - Army	Doc Data Sht
Air Force Scientific Adviser	1
Director Trials	1
Aeronautical and Maritime Research Laboratory	
Director, Aeronautical and Maritime Research Laboratory	1
Electronics and Surveillance Research Laboratory	
Director, Electronics and Surveillance Research Laboratory	Doc Data Sht
Chief, Surveillance Systems Division	1
Research Leader, Imagery Systems	1
Head, Image Analysis & Exploitation	1
Dr Alan Rye	1
David J. Robinson	1
Dr Nicholas J. Redding	4
Dr David J. Crisp	4
David I. Kettler	1
Dr Tim Payne	1
Dr Nick J. S. Stacy	1
Rodney Smith	1
DSTO Library and Archives	
Library Fishermans Bend	Doc Data Sht

Library Maribyrnong	Doc Data Sht
Library Edinburgh	1
Australian Archives	1
Library, MOD, Pyrmont	Doc Data Sht
US Defense Technical Information Center	2
UK Defence Research Information Centre	2
Canada Defence Scientific Information Service	1
NZ Defence Information Centre	1
National Library of Australia	1
Capability Systems Staff	
Director General Maritime Development	Doc Data Sht
Director General Land Development	1
Director General Aerospace Development	Doc Data Sht
Knowledge Staff	
Director General Command, Control, Communications and Computers (DGC4)	Doc Data Sht
Army	
Stuart Schnaars, ABCA Standardisation Officer, Tobruk Barracks, Puckapunyal, 3662	4
SO (Science), Deployable Joint Force Headquarters (DJFHQ) (L), MILPO Gallipoli Barracks, Enoggera QLD 4052	Doc Data Sht
Intelligence Program	
DGSTA Defence Intelligence Organisation	1
Manager, Information Centre, Defence Intelligence Organisation	1
Corporater Support Program	
Library Manager, DLS-Canberra	1
UNIVERSITIES AND COLLEGES	
Australian Defence Force Academy, Library	1
Australian Defence Force Academy, Head of Aerospace and Mechanical Engineering	1
Serials Section (M list), Deakin University Library, Geelong, 3217	1
Hargrave Library, Monash University	Doc Data Sht
Librarian, Flinders University	1
OTHER ORGANISATIONS	
NASA (Canberra)	1

AusInfo	1
State Library of South Australia	1

OUTSIDE AUSTRALIA

Abstracting and Information Organisations

Library, Chemical Abstracts Reference Service	1
Engineering Societies Library, US	1
Materials Information, Cambridge Scientific Abstracts, US	1
Documents Librarian, The Center for Research Libraries, US	1

Information Exchange Agreement Partners

Acquisitions Unit, Science Reference and Information Service, UK	1
Library - Exchange Desk, National Institute of Standards and Technology, US	1

SPARES

DSTO Salisbury Research Library	5

Total number of copies:	58

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION DOCUMENT CONTROL DATA				1. CAVEAT/PRIVACY MARKING
2. TITLE Implementation of a Fast Algorithm for Segmenting SAR Imagery			3. SECURITY CLASSIFICATION Document (U) Title (U) Abstract (U)	
4. AUTHORS David J. Robinson, Nicholas J. Redding and David J. Crisp			5. CORPORATE AUTHOR Electronics and Surveillance Research Laboratory PO Box 1500 Edinburgh, South Australia, Australia 5111	
6a. DSTO NUMBER DSTO–TR–1242	6b. AR NUMBER 012-076		6c. TYPE OF REPORT Technical Report	7. DOCUMENT DATE January, 2002
8. FILE NUMBER B 9505-21-184	9. TASK NUMBER ARM 97/261	10. SPONSOR DGAD	11. No OF PAGES 34	12. No OF REFS 11
13. URL OF ELECTRONIC VERSION http://www.dsto.defence.gov.au/corporate/reports/DSTO–TR–1242.pdf			14. RELEASE AUTHORITY Chief, Surveillance Systems Division	
15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT *Approved For Public Release* OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SOUTH AUSTRALIA 5111				
16. DELIBERATE ANNOUNCEMENT No Limitations				
17. CITATION IN OTHER DOCUMENTS No Limitations				
18. DEFTEST DESCRIPTORS synthetic aperture radar image analysis segmentation				

19. ABSTRACT

This report gives a detailed presentation of the implementation of a new fast algorithm for image segmentation. The original motivation for development of the algorithm was the segmentation of synthetic aperture radar (SAR) imagery into homogeneous regions for target detection in the Analysts' Detection Support System. However, the algorithm is a general one based upon Mumford-Shah functionals, and there is no technical reason why it could not also be used for other imaging modalities, including multiband imagery. The algorithm has computational complexity on the order of the Fast Fourier Transform, the benchmark for fast algorithms.

TECHNICAL REPORT DSTO-TR-1242 AR-012-076 JANUARY 2002

DEPARTMENT OF DEFENCE
DEFENCE SCIENCE & TECHNOLOGY ORGANISATION
DSTO

ELECTRONICS AND SURVEILLANCE RESEARCH LABORATORY
PO BOX 1500 EDINBURGH SOUTH AUSTRALIA, 5111
AUSTRALIA, TELEPHONE (08) 8259 5555